Code Generation

Translator from Mini-Pascal to simplified C

Petri Kallio, 012843064

June 1, 2018

---

1 How to build and use the translator

The project is made into a MonoDevelop solution. It contains three MonoDevelop projects:

1. `Compiler` contains the source code of all the different components in the compiler and simplified-C-synthesizer and is meant to be compiled into a `Compiler.dll` dynamic library file that the other two projects use.
2. `Translator` contains the source code of the main program and is meant to be compiled into an `Translator.exe` executable file. It needs to be in the same folder with the `Compiler.dll` to be executed.
3. `CompilerTests` contains the unit tests for the compiler and they also need `Compiler.dll` to be run. The tests use Nunit framework.

To run the interpreter (in Windows), make sure the library and the executable are in the same directory and then execute the program by typing

```
Interpreter.exe source-file.ext [target-file.c]
```

where `source-file.ext` is the path to the source file you want to translate. The second parameter is optional and it is the target file that will contain the translated simplified C code. If no second argument is given, the translation will be printed out to standard output.

To run the interpreter in Linux, make sure you have Mono installed and type

```
mono Interpreter.exe source-file.ext [target-file.c]
```

where `source-file.ext` is again the path to the source file you want to compile and translate.

You can also run it from MonoDevelop IDE.

2 Mini-Pascal translator architecture

In its highest level the translator's architecture is divided into four components: main program, compiler frontend, synthesizer and I/O.

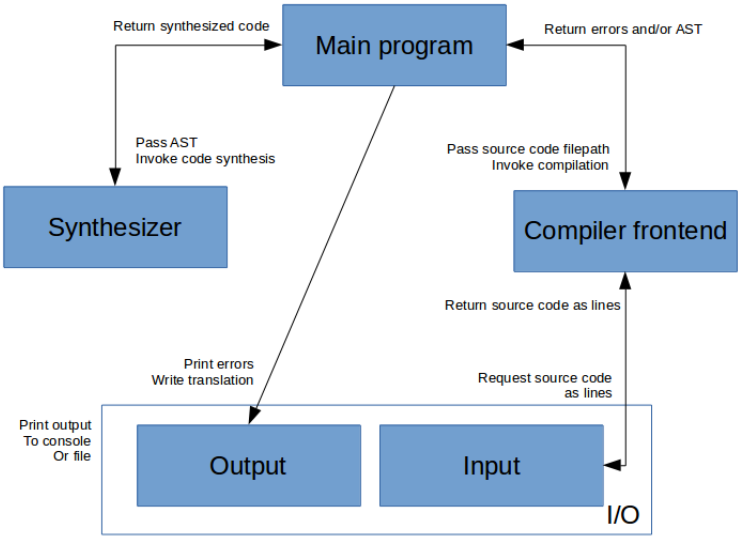

Figure 1: The overall architecture of the translator.

The purpose of the main program is of course to control all the needed resources, to inform the user about errors during compilation, to make the decision of whether to perform the code synthesis or not and inform the user of the completion of the process.

The purpose of the I/O is to hide the source file read operations from the compiler frontend and to provide output functionality for the main program.

The purpose of the frontend is to analyze the source code and then provide any errors found during the analysis. The compilation, if executed without errors, provides the main program with an abstract syntax tree (AST) of the program.

The purpose of the synthesizer is to make a synthesis of the program provided as AST by the main program to simplified C.

Compiler frontend

The compiler frontend's architecture is divided into three components in its highest level: scanner, parser and semantic analyzer. These three provide three different interleaved phases of the frontend compilation.

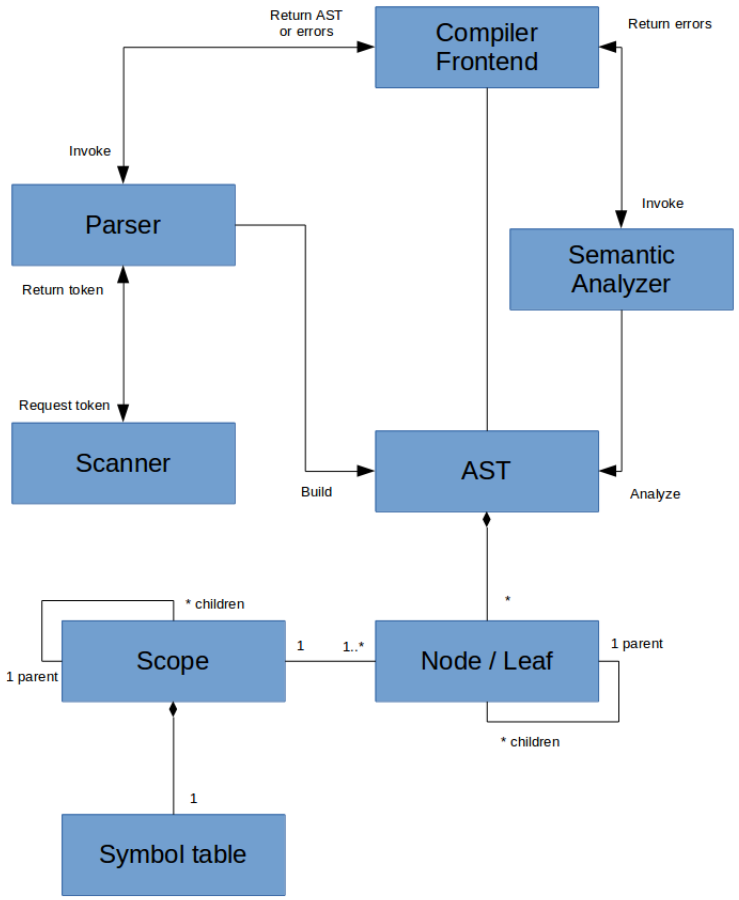

Figure 2: The overall architecture of the compiler frontend.

The compilation is done in three phases that are divided into two passes. During the first pass the scanner and the parser cooperate to perform lexical and syntactical analysis of the source code. The pass is made over the original source code. As a product they supply the main program with either a list of lexical, syntactical (and semantical) errors found during the analysis, or a complete AST of the source code. The parser also builds the AST nodes' scopes and their symbol tables as well as provides semantical errors in case a variable identifier is declared more than once inside a scope. The second pass (and the third phase) is the static semantical analysis of the AST. This phase is performed only if the first pass was error-free.

The parser has the control during the first pass. It requests one token at a time from the scanner and performs its analysis and builds the AST on its way. The parser is implemented using recursive-descent parsing.

During semantic analysis the AST is inspected to enforce static semantic rules and it's implemented using the **Visitor** design pattern.

Synthesizer

If the compilation finishes without errors, the AST is passed to the synthesizer that translates the AST to simplified C. Like the semantic analyzer, the synthesizer is also implemented using the **Visitor** design pattern.

3 Language implementation-level decisions

The following decisions were made:

1. a function's last statement's all code paths must return a value
2. an array as a function's or procedure's parameter can have a size expression inside the brackets, but it has no effect inside the function and it is not checked when the function is called
3. a string as a function's or procedure's parameter must be passed by reference
4. an array as a function's or procedure's parameter must be passed by reference
5. the evaluation of expression's is from left to right
6. a simple type variable must have a value assigned to it before using it in an expression
7. if a predefined identifier is used as a variable identifier the meaning of that id is the meaning of the variable in the scope it is declared in, as well as in the child scopes

4 Lexical analysis (scanning phase)

The scanner is implemented as a deterministic finite automaton (DFA) which uses the combined regular expressions of all the valid tokens to make decisions while scanning. In this section the scanner's functionality is explained.

Valid Mini-Pascal token patterns

The scanner identifies tokens from the character stream by identifying valid token patterns.

The valid token patterns are given as regular expressions. In addition to the common symbols used in regular expressions, here are explanations of the other symbols used in the regular definitions:

`[0-9aqz]` matches any character in the given range '0' to '9' and characters 'a', 'q' and 'z'

`[^0-9aqz]` matches any character EXCEPT the range '0' to '9' or characters 'a', 'q' and 'z', including a linebreak

`\n` a linebreak

`\(` the character '(', not the regular expression grouping parenthesis

`a?` the pattern is optional, meaning it is repeated 0 or 1 times

`a+` the pattern is repeated 1 or more times

All the regular expression quantifiers are superscripted, so `*`* would mean 0 or more `*`-symbols and `?`? would mean 0 or 1 `?`-symbols. The regular expressions for different tokens:

```
    special symbol ::= <(=|>)? | >=? | = | ; | \( | \) | \[ | \]
                     | * | / | + | - | . | % | , | :=?
           integer ::= [0-9]+
              real ::= [0-9]+.[0-9]+(e(-|+)?[0-9]+)?
        identifier ::= [a-zA-Z]([a-zA-Z0-9]|_)*
            string ::= "[^\n]*"
multi-line comment ::= {*([^*]|*+[^*})*+}
```

By combining the overlap of integers and reals, we get the following regular expression:

```
integer or real ::= [0-9]+(.[0-9]+(e(-|+)?[0-9]+)?)?
```

Altogether we get the following combined regular expression for valid token patterns:

```
mini-pascal token ::= <(=|>)? | >=? | = | ; | \( | \) | \[ | \]
                    | * | / | + | - | . | % | , | :=?
                    | [0-9]+(.[0-9]+(e(-|+)?[0-9]+)?)?
                    | [a-zA-Z]([a-zA-Z0-9]|_)* | "[^\n]*"
                    | {*([^*]|*+[^*})*+}
```

Error handling

The error handling in the scanner is implemented in the following manner:

1. If the token is a string literal and it spans multiple lines, it reports an error.
2. If the token is a string literal and the end of file is reached while scanning, it reports an error.
3. If the token is invalid, it reports an error.
4. If a real number's form is wrong it reports an error.

Other remarks

Some remarks on the functionality of the scanner:

1. Because the scanner is implemented so that the maximal munch rule is obeyed, identifiers and integers must be delimited by non-identifier and non-numeric characters respectively.

2. Keywords are identified as respective tokens by screening identifier tokens; the scanner first scans an identifier and then compares the identifier to a dictionary of keyword to determine whether it is a keyword or a variable identifier.

3. Comments and whitespaces are screened as well; whenever a comment or a whitespace is read from the character stream, it is passed by and not returned as a token.

4. The tokens supplied by the scanner contain information about

   - the token's type,
   - its corresponding value and
   - its position in the source code (row and column).

   The tokens and their types are maintained during the whole execution of the program, attached to their corresponding nodes in the AST. This helps creating valuable information to the user when printing errors. Other information is represented using the `TokenType` enumerator as well, such as the evaluation value of an expression.

5 Syntax analysis (parsing phase)

The parser is implemented as a top-down recursive-descent parser. In this section the parser's functionality is explained.

LL-grammar

Because the parser is implemented as a top-down parser, it needs a LL-grammar to make decisions while parsing. Below is an LL-grammar that defines the Mini-Pascal programming language accompanied by the expectation sets in curly braces that are needed to choose the right rule while parsing.

$$
\begin{aligned}
\langle \text{program} \rangle &\rightarrow \textbf{"program"}\ \langle \text{id} \rangle\ \textbf{";"}\ \langle \text{program-mid} \rangle\ \langle \text{block} \rangle\ \textbf{"."}\ \{\texttt{"program"}\} \\
\langle \text{program-mid} \rangle &\rightarrow \langle \text{procedure} \rangle\ \langle \text{program-mid} \rangle\ \ \{\texttt{"procedure"}\} \\
&\rightarrow \langle \text{function} \rangle\ \langle \text{program-mid} \rangle\ \ \{\texttt{"function"}\} \\
&\rightarrow \varepsilon\ \{\texttt{"begin"}\} \\
\langle \text{procedure} \rangle &\rightarrow \textbf{"procedure"}\ \langle \text{id} \rangle\ \textbf{"("}\ \langle \text{parameters} \rangle\ \textbf{")"}\ \textbf{";"}\ \langle \text{block} \rangle\ \textbf{";"}\ \ \{\texttt{"procedure"}\} \\
\langle \text{function} \rangle &\rightarrow \textbf{"function"}\ \langle \text{id} \rangle\ \textbf{"("}\ \langle \text{parameters} \rangle\ \textbf{")"}\ \textbf{":"}\ \langle \text{type} \rangle\ \textbf{";"}\ \langle \text{block} \rangle\ \textbf{";"}\ \ \{\texttt{"function"}\} \\
\langle \text{var-declaration} \rangle &\rightarrow \textbf{"var"}\ \langle \text{id} \rangle\ \langle \text{var-declaration-mid} \rangle\ \textbf{":"}\ \langle \text{type} \rangle\ \ \{\texttt{"var"}\} \\
\langle \text{var-declaration-mid} \rangle &\rightarrow \textbf{","}\ \langle \text{id} \rangle\ \langle \text{var-declaration-mid} \rangle\ \ \{\texttt{","}\}
\end{aligned}
$$

$\rightarrow \varepsilon$ {":"}

⟨parameters⟩ → ⟨param⟩ ⟨param-tail⟩ {"var", ID}

$\rightarrow \varepsilon$ {")"}

⟨param⟩ → **"var"** ⟨id⟩ **":"** ⟨type⟩ ⟨param-tail⟩ {"var"}

→ ⟨id⟩ **":"** ⟨type⟩ ⟨param-tail⟩ {ID}

⟨param-tail⟩ → **","** ⟨param⟩ {","}

$\rightarrow \varepsilon$ {")"}

⟨type⟩ → ⟨simple-type⟩ {"integer", "string", "Boolean", "real"}

→ ⟨array-type⟩ {"array"}

⟨simple-type⟩ → **"integer"** {"integer"}

→ **"string"** {"string"}

→ **"Boolean"** {"Boolean"}

→ **"real"** {"real"}

⟨array-type⟩ → **"array"** **"["** ⟨array-type-mid⟩ **"]"** **"of"** ⟨simple-type⟩ {"array"}

⟨array-type-mid⟩ → ⟨expression⟩ {"+", "-", ID, STRING, INTEGER, REAL, "(", "not"}

$\rightarrow \varepsilon$ {"]"}

⟨block⟩ → **"begin"** ⟨statement⟩ ⟨block-mid⟩ ⟨block-mid-tail⟩ **"end"** {"begin"}

⟨block-mid⟩ → **";"** ⟨statement⟩ ⟨block-mid⟩ {";"}

$\rightarrow \varepsilon$ {";", "end"}

⟨block-mid-tail⟩ → **";"** {";"}

$\rightarrow \varepsilon$ {"end"}

⟨statement⟩ → ⟨simple-statement⟩ {ID, "return", "read", "writeln", "assert"}

→ ⟨structured-statement⟩ {"begin", "if", "while"}

→ ⟨var-declaration⟩ {"var"}

⟨simple-statement⟩ → ⟨id⟩ ⟨simple-statement-id-tail⟩ {ID}

→ ⟨return-statement⟩ {"return"}

→ ⟨read-statement⟩ {"read"}

→ ⟨write-statement⟩ {"writeln"}

→ ⟨assert-statement⟩ {"assert"}

⟨simple-statement-id-tail⟩ → ⟨assign-statement-tail⟩ {":="}

→ **"["** ⟨expr⟩ **"]"** ⟨assign-statement-tail⟩ {"["}

→ ⟨call-tail⟩ {"("}

⟨assign-statement-tail⟩ → **":="** ⟨expr⟩ {":="}

⟨call-tail⟩ → **"("** ⟨arguments⟩ **")"** {"("}

⟨arguments⟩ → ⟨expr⟩ ⟨arguments-tail⟩ {"+", "-", ID, STRING, INTEGER, REAL, "(", "not"}

$\rightarrow \varepsilon$ {")"}

⟨arguments-tail⟩ → **","** ⟨expr⟩ ⟨arguments-tail⟩ {","}

$\rightarrow \varepsilon$ {")"}

⟨return-statement⟩ → **"return"** ⟨return-statement-tail⟩ {"return"}

⟨return-statement-tail⟩ → ⟨expr⟩ {"+", "-", ID, STRING, INTEGER, REAL, "(", "not"}

$\rightarrow \varepsilon$ {"else", ";", "end"}

⟨read-statement⟩ → **"read"** **"("** ⟨variable⟩ ⟨read-statement-tail⟩ **")"** {"read"}

⟨read-statement-tail⟩ → **","** ⟨variable⟩ ⟨read-statement-tail⟩ {","}

$\rightarrow \varepsilon$ {")"}

⟨write-statement⟩ → **"writeln"** **"("** ⟨arguments⟩ **")"** {"writeln"}

⟨assert-statement⟩ → **"assert"** **"("** ⟨expr⟩ **")"** {"assert"}

⟨structured-statement⟩ → ⟨block⟩ {"begin"}

→ ⟨if-statement⟩ {"if"}

→ ⟨while-statement⟩ {"while"}

⟨if-statement⟩ → **"if"** ⟨expr⟩ **"then"** ⟨statement⟩ ⟨else-statement⟩ {"if"}

⟨else-statement⟩ → **"else"** ⟨statement⟩ {"else"}

→ ε {";", "end"}

⟨while-statement⟩ → **"while"** ⟨expr⟩ **"do"** ⟨statement⟩ {"while"}

⟨expr⟩ → ⟨simple-expr⟩ ⟨expr-tail⟩ {"+", "-", ID, STRING, INTEGER, REAL, "(", "not"}

⟨expr-tail⟩ → ⟨relational-operator⟩ ⟨simple-expr⟩ {"=", "<>", "<", "<=", ">=", ">"}

→ ε {")", ","}

⟨simple-expr⟩ → ⟨sign⟩ ⟨term⟩ ⟨simple-expr-tail⟩ {"+", "-"}

→ ⟨term⟩ ⟨simple-expr-tail⟩ {ID, STRING, INTEGER, REAL, "(", "not"}

⟨simple-expr-tail⟩ → ⟨adding-operator⟩ ⟨term⟩ ⟨simple-expr-tail⟩ {"+", "-", "or"}

→ ε {"]", "end", ";", ",", ")", "then", "else", "do"}

⟨term⟩ → ⟨factor⟩ ⟨term-tail⟩ {ID, STRING, INTEGER, REAL, "(", "not"}

⟨term-tail⟩ → ⟨multiplying-operator⟩ ⟨factor⟩ ⟨term-tail⟩ {"*", "/", "%","and"}

→ ε {"]", "end", ";", ",", ")", "then", "else", "do"}

⟨factor⟩ → ⟨factor-main⟩ ⟨factor-tail⟩ {ID, STRING, INTEGER, REAL, "(", "not"}

⟨factor-main⟩ → ⟨id⟩ ⟨factor-id-tail⟩ {ID}

→ ⟨literal⟩ {STRING, INTEGER, REAL, BOOLEAN}

→ **"("** ⟨expr⟩ **")"** {"("}

→ **"not"** ⟨factor⟩ {"not"}

⟨factor-tail⟩ → **"."** **"size"** {"."}

→ ε {"]", "end", ";", ",", ")", "then", "else", "do"}

⟨factor-id-tail⟩ → ⟨call-tail⟩ {"("}

→ ⟨variable-tail⟩ {"["}

→ ε {".", "]", "end", ";", ",", ")", "then", "else", "do"}

⟨variable⟩ → ⟨variable-id⟩ ⟨variable-tail⟩ {ID}

⟨variable-tail⟩ → **"["** ⟨expr⟩ **"]"** {"["}

→ ε {":=", ",", ")"}

⟨relational-operator⟩ → **"="** {"="}

→ **"<>"** {"<>"}

→ **"<"** {"<"}

→ **"<="** {"<="}

→ **">="** {">="}

→ **">"** {">"}

⟨sign⟩ → **"+"** {"+"}

→ **"-"** {"-"}

⟨negation⟩ → **"not"** {"not"}

⟨adding-operator⟩ → **"+"** {"+"}

$$
\begin{aligned}
&\rightarrow \text{"-"} \quad \{\texttt{"-"}\} \\
&\rightarrow \textbf{"or"} \quad \{\texttt{"or"}\} \\
\langle \text{multiplying-operator} \rangle &\rightarrow \text{"*"} \quad \{\texttt{"*"}\} \\
&\rightarrow \text{"/"} \quad \{\texttt{"/"}\} \\
&\rightarrow \text{"\%"} \quad \{\texttt{"\%"}\} \\
&\rightarrow \textbf{"and"} \quad \{\texttt{"and"}\}
\end{aligned}
$$

The symbols inside angles are non-terminal symbols. The capitalized symbols in the expectation set are variable literals (string, integer, real and variable identifier literals). The ε-character depicts an empty token.

Remaining ambiguities in the grammar

There is one ambiguity that could not be removed while transforming the grammar. A semicolon after the last statement of a block is optional. This means that after each statement we have to peek the next symbol that comes after the semicolon. If it's the `end` keyword, we choose the rule

$$\langle \text{block-mid} \rangle \rightarrow \varepsilon$$

and if it's not, we choose the rule

$$\langle \text{block-mid} \rangle \rightarrow \text{";"} \ \langle \text{statement} \rangle \ \langle \text{block-mid} \rangle.$$

Building the AST and the scopes

As mentioned before, the parser is also responsible for building the AST and its nodes' scopes while parsing. The scope is changed every time one of the following occurs:

1. a program is parsed,
2. a function is parsed,
3. a procedure is parsed or
4. a block is parsed.

Since the parsing begins with the program node, its scope will become the root scope of the program. After this, when we start parsing a function, procedure or block, we declare and enter a new scope whose parent will be the scope we entered this scope from. after the parsing of this said function, procedure or block, we re-enter its parent scope. This way, when we search a variable from the scope tree, we can look for it from the current scope as well as from all of its ancestral scopes, but not from its sibling scopes or its ancestors' sibling scopes.

Error handling

The error handling in the parser is implemented so that the parsing method throws an exception when faced with an invalid token. The catching method then disables the building of the AST and fastforwards to a safe point from which to continue parsing. This means that even one error while parsing will prevent the safe interpretation of the program.

Other remarks

1. Since some of the keywords in Mini-Pascal are defined as predefined identifiers, when the scanner returns a token whose type is not a variable identifier the parser tries to convert the token into an identifier. If it is a predefined id and passes the parser's error checking, it is converted to and subsequently used as an identifier. This means that, for example, if an identifier `true` is parsed it can no more be used as a keyword.

6 Semantic analysis

The semantic analysis phase is implemented by using the **Visitor** design pattern. All the nodes of the AST are required to implement the `Accept(Visitor visitor)` method defined in the `ISyntaxTreeNode` interface.

The semantic analyzer creates an instance of `StatementCheckVisitor`. Then the AST's root node's `Accept` method is called with the `StatementCheckVisitor` as it's argument. The visitor then checks the whole AST in a depth-first fashion when each node calls the `Accept` methods of their children.

The other visitor required in this phase is the `TypeCheckingVisitor`, which checks the nodes' expressions' evaluation types. The `StatementCheckVisitor` calls a node's `Accept` method with a `TypeCheckingVisitor` as its argument whenever needed.

Semantic rules

The rules enforced while doing semantic analysis to the AST nodes are the following:

Function

- the last statement's all code paths return a value
  - the last statement cannot be a while loop
  - the last statement cannot be an if-then statement
  - if the last statement is and if-then-else statement, both then and else branches' all sub-branches must return a value
- all the return statements must return a value that matches the function's return type

Parameters

- parameter identifiers are unique
- an array parameter is always a pointer
- a string parameter is always a pointer

Variable declaration

- the identifiers are unique within the same scope

Array type

- the size expression evaluates to integer

Array access

- the variable identifier is declared
- the variable identifier points to an array
- the indexing expression evaluates to an integer

Assignment

- the variable has been declared before within the same scope or in one of the ancestral scopes
- the variable's and expression's types match

Function/procedure call

- variable identifier points to a function or a procedure
- arguments' types match parameters' types

Return-statement

- statement's type matches the function's/procedure's return type

Read-statement

- all variables given as arguments are declared
- none of the arguments points to a boolean variable
- none of the arguments points to an array variable

Assert-statement

- the expression evaluates to a boolean value

If-statement

- the condition evaluates to a boolean value

While-statement

- the condition evaluates to a boolean value

Expression

- the first and last operands match and are supported by the operation

Simple expression

- the first operand supports sign operation
- the first and last operands match and are supported by the operation

Simple expression tail

- the operand supports the operation
- the operand's and the tail's evaluation type match and are supported by the tail's operation

Term

- the factor and term tail match and are supported by the tail's operation

Term tail

- the factor and is supported by the operation
- the factor and the child term tail match and are supported by the child term tail's operation

Factor

- the factor-tail supports the factor-main

Factor-main

- the variable identifier is declared
- the variable identifier is assigned
- the variable identifier's evaluation type is supported by the factor-id-tail

Boolean negation

- the factor evaluates to a boolean value

Error handling

Whenever a static semantic error is encountered, it is reported.

7 Abstract syntax tree (AST)

The AST is built using the Composite design pattern. An object of class `SyntaxTree` contains a pointer to its root node. All the nodes inherit the class `SyntaxTreeNode`, either directly or indirectly through another class.

Common properties to all the nodes are:

`Token` accessor The token that this node represents in the source

`Scope` accessor	The scope that this node belongs to
`Location` accessor	The temporary location this nodes evaluation is held at in the translation
`Label` accessor	The label in the translation that points to this node's code
`Accept` method	Accepts an `INodeVisitor`

The node hierarchy is shown in the figures that follow:

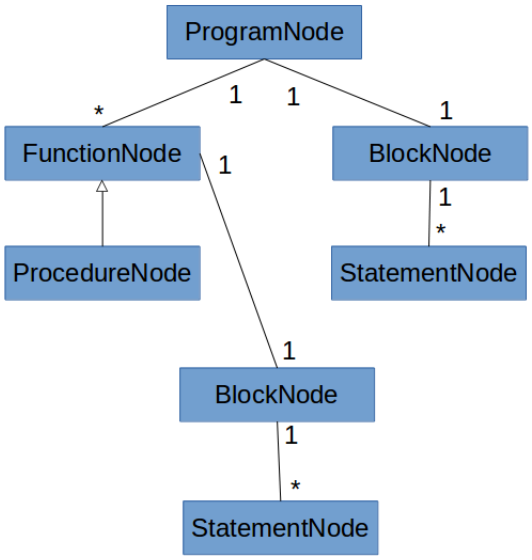

Figure 3: The program node is the root of the AST and connects to the function and procedure nodes, as well as to the main block of the program.

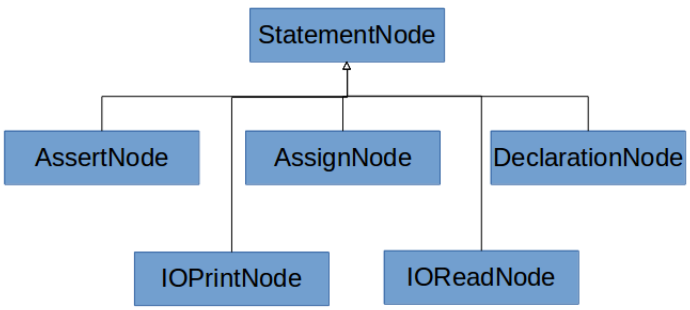

Figure 4: First set of the nodes that inherit the statement node.

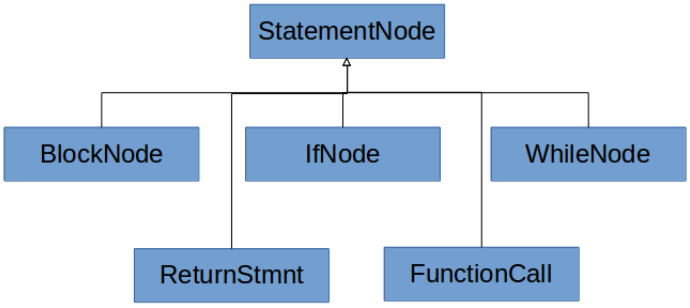

Figure 5: Second set of the nodes that inherit the statement node.

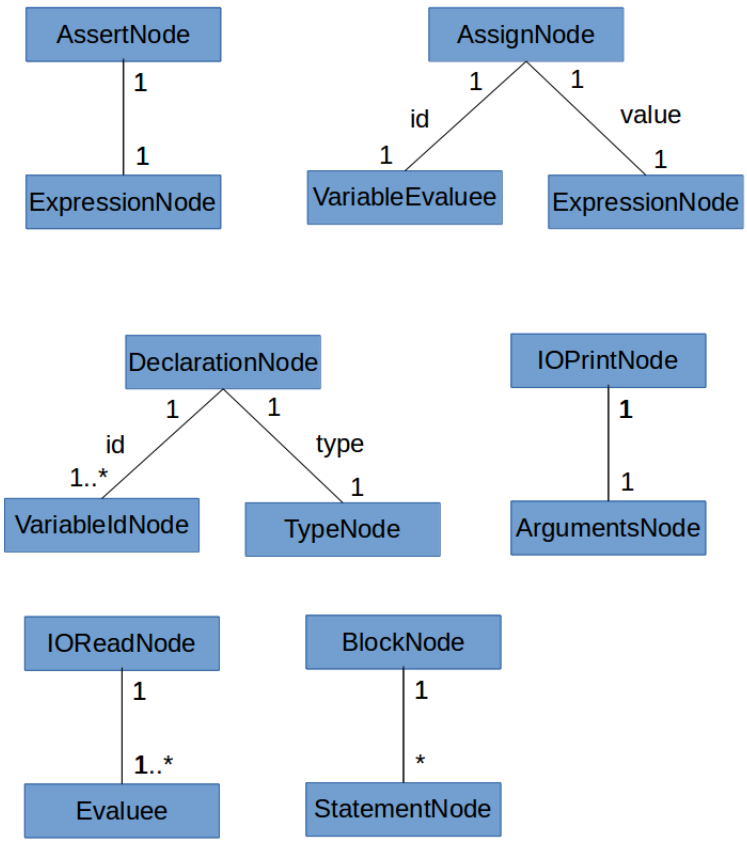

Figure 6: The AST's for assert, assign, declaration, IO-print, IO-read and block nodes.

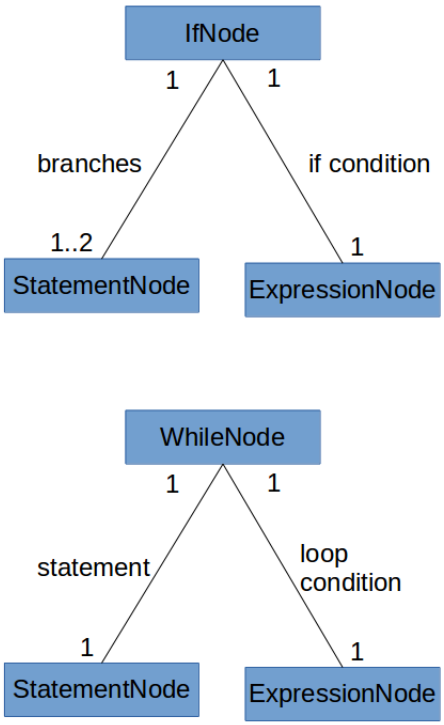

Figure 7: The AST's for if and while nodes.

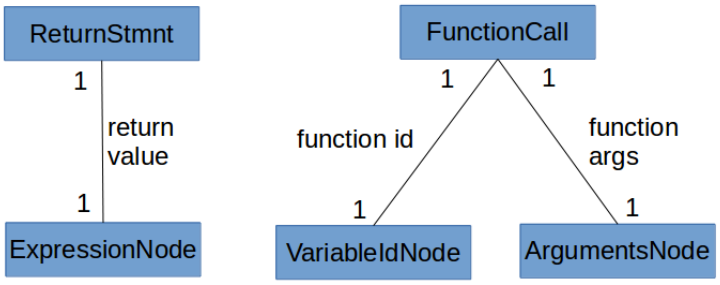

Figure 8: The AST's for return statement and function call nodes.

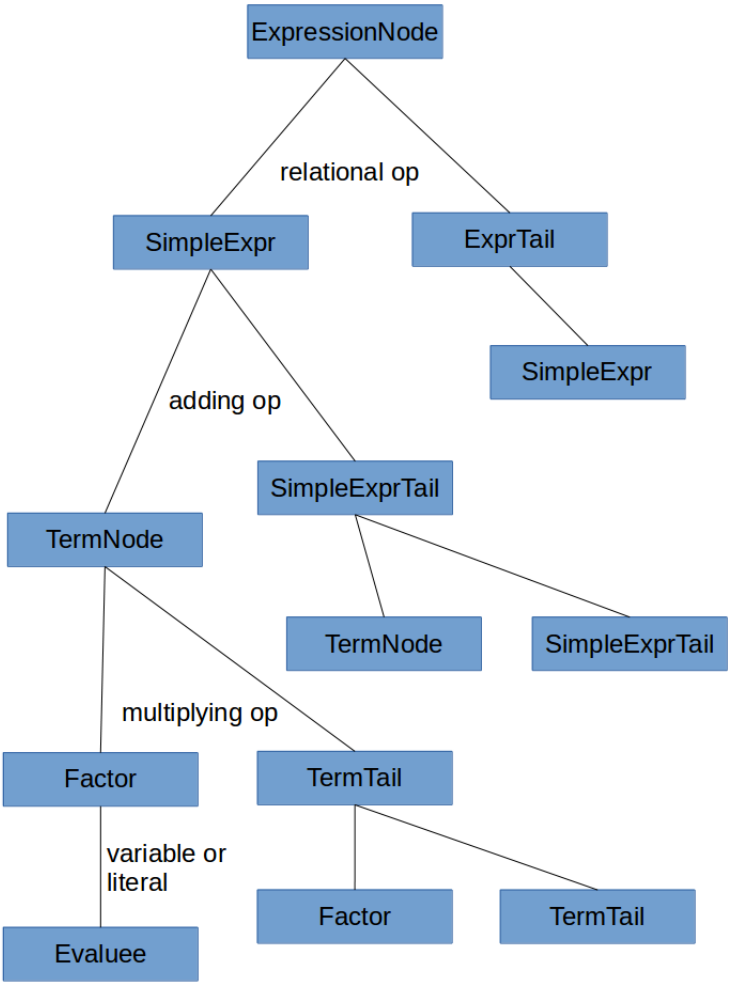

Figure 9: The AST's for expression node. The precedence of different operations can be easily seen: multiplying operations first, then adding operations and lastly the relational operation.

8 Synthesis

The synthesis phase, like the semantic analysis phase, is implemented using the **Visitor** design pattern. The AST's root node's `Accept` method is called with an `CSynthesisVisitor` as its argument. The execution is then performed in a depth-first fashion to each node's children.

Libraries

The use of libraries is kept to absolute minimum. Each Simplified C translation includes `stdio.h` header file for standard input and output and `stdlib.h` for allocating and deallocating memory for arrays.

Types

The types are converted as follow:

Mini-Pascal type	**C type**
`integer`	`int`
`real`	`float`
`boolean`	`int` (false if zero, true otherwise)
`string`	`char*`
`array[] of integer`	`int*`
`array[] of real`	`float*`
`array[] of boolean`	`int*`
`array[] of string`	`char**`

Helper functions

Helper functions are added to each translated file that are used to

- count the length of a string,
- load an element from an array,
- insert an element to an array,
- concatenate two strings,
- compare two strings and
- print to standard output.

Naming convention

All the Mini-Pascal's identifiers could be used as such in C. Yet there is a danger that the user has declared such an identifier in his/her Mini-Pascal program that it is not accepted in C. That's why all the identifiers in C are prepended with underscore. These will not collide with any of the Mini-Pascal's other identifiers since they must begin with a letter.

Arrays

Arrays are created using dynamic allocation (`malloc`). If the size of the array in the source code is n, an array of $n + 1$ elements is allocated. The size of the array is then cast to the array's type and saved as the first element of the array. The remaining n slots are used so that if a reference is made to index x, the element is retrieved from index $x + 1$ of the array. Arrays size is checked every time an insertion or load is made to avoid loading or inserting out of the array's bounds.

Scopes

The original scopes are flattened so that only one scope for each function and procedure remains. If redeclarations are made in the source program, in addition to prepending an underscore to the identifier, the underscore will be followed by the original scope's redeclaration number. This will not collide with any other identifier in Mini-Pascal, since they cannot begin with a number.

Error handling

A global integer variable `ERRORCODE` and a scope specific `error` label are used to handle errors. If an error is encountered (invalid input, index out of bounds, failed assertion) the error code is set to 1 and the execution jumps to the scope's error label. Subsequently, every parent scope will check the error code and jump to their error label if the error code is not zero.

9 Testing

In addition to the case specific contextual testing during the development of each feature, the testing of the Mini-Pascal translator is implemented by extensive unit testing of different parts of the compiler frontend and synthesizer. The unit tests are included within the solution as their own project named `CompilerTests`.

All the different test files contain tests both for valid and invalid sources.

10 Shortcomings

1. Some of the array operations in the synthesized C code result to an error.
2. The temporary variables used during the synthesis are not properly freed and a lot of temporary variables are created in vain in the target code.
3. Refactoring is not done to the synthesizer, so the code's many times quite ugly.
4. The synthesis of an IO-read operation for an array doesn't work correctly. The read is made to the arrays address, overwriting the information about the size of the array.
5. the dynamic allocations are not freed, it caused an error and I didn't have time to fix it
6. the code commenting is partly undone and partly out of date